Advanced Java Programming

Week 6 Topic Outline

Introduction to Generics

- What are the <> brackets we see in these collection class definitions?
- Defining a generic method
- Defining a generic class
- **Terminology**: *Raw type*, *Generic type*, *Type parameter*, *Type argument*
- Conventional lettering:

Letter	Meaning
E	Element - an element of a list
K	Key - the key used in a (key, value) pair
V	Value - the value used in a (key, value) pair
N	Number - a number
T	Type - a generic type; usually used in class definitions

Bounded Type Parameters

- Can do more with generic types if we could assume some common shared feature
- Can enforce an *upper bound* which the argument must subclass
- Syntax: `<E extends Foo>` or `<E extends Foo & Bar>`

Wildcards

- If we make no explicit references to the data (e.g. no variables of type `E`), we can use wildcards.
- `public static <E> void foo(List<E> list)` ⟹ `public static void foo(List<?> list)`
- `public static <E extends Bar> void foo(List<E> list)` ⟹ `public static void foo(List<? extends Bar> list)`

Type Erasure & Reification

- Java did not originally have generics (added as backwards-compatible)
- *Reified (v.) - Regarding something abstract as if it were a concrete, material thing.*
- We say that types are *reifiable* if we can access them at runtime, and *non-reifiable* if they are erased after type checking.
- Generic types are non-reifiable (they are erased). They only apply during the type-checking phase of compilation.

Runtime transformations:

- `<T>` is translated to `Object`
- `<T extends Foo>` is translated to `Foo`

Consequences:

- Cannot instantiate generic types with primitive types
    - `<T>` is translated to `Object`, so `T` must be an object reference
- Cannot instantiate generic types
- Cannot create arrays of generic types
- Cannot declare static fields with generic types
- Cannot use `instanceof` with generic types (because `instanceof` is a runtime check)
- Cannot overload method where formal parameter types erase to same raw type

Reflection

- Java's *reflection* API lets code inspect itself
- Pros: allows you to instantiate generic types and arrays
- Cons: runs slowly and reflective code is difficult to read

Subtypes

- `Bar` is a *subtype* of `Foo` iff a value of type `Bar` can be used anywhere a value of type `Foo` is used. In that case, `Foo` is a *super-type* of `Bar`.
- If `Bar` subclasses `Foo` then `Bar` is a subtype of `Foo`
- Subclass relationship is more specific than subtype relationship (it also implies inheritance etc.)
- Notation (not Java syntax): `Bar <: Foo` and `Foo >: Bar`

Invariance

- `Foo` is *invariant* iff, given `Foo<Bar>` and `Foo<Baz>`, neither `Bar <: Baz` nor `Baz <: Bar` implies anything about the relationship between `Foo<Bar>` and `Foo<Baz>`.
- Generic classes (like ArrayLists) are invariant
    - Given `Integer <: Number`
    - `ArrayList<Integer>` and `ArrayList<Number>` cannot interact:
        - Cannot pass an `ArrayList<Integer>` to a method that takes an `ArrayList<Number>`
        - Cannot reference an `ArrayList<Integer>` with an `ArrayList<Number>` pointer

Covariance

- `Foo` is *covariant* iff, given `Foo<Bar>` and `Foo<Baz>`, `Bar <: Baz` → `Foo<Bar> <: Foo<Baz>`.
- Arrays in Java are covariant. Can reference an `Integer[]` with a `Number[]` pointer.
- Covariance is manifested in Java as a wildcard upper bound:
    - `public boolean addAll(Collection<? extends E> c)`

Contra-variance

- `Foo` is *contra-variant* iff, given `Foo<Bar>` and `Foo<Baz>`, `Bar <: Baz` → `Foo<Bar> >: Foo<Baz>`.
- Contra-variance is manifested in Java as a wildcard *lower bound*:
    - `public void copyTo(Collection<? super E> c)`

Producers Extend, Consumers Super (PECS)

- When do we want covariance? When we are supplying / *producing* values
    - Something that can produce integers can be used when we need something that can produce numbers.
- When do we want contra-variance? When we are *consuming* values
    - Something that can consume numbers can be used when we need something that can consume integers.